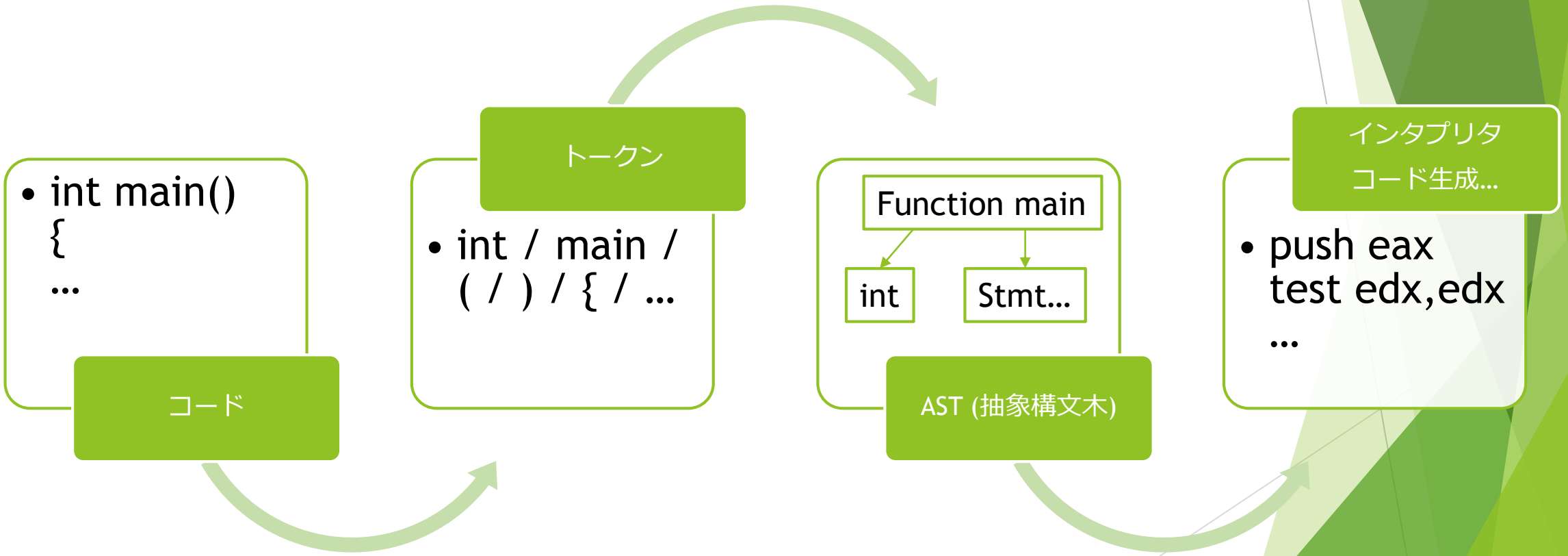
言語処理系分科会
第 1 回 - lex, yacc

semiexp

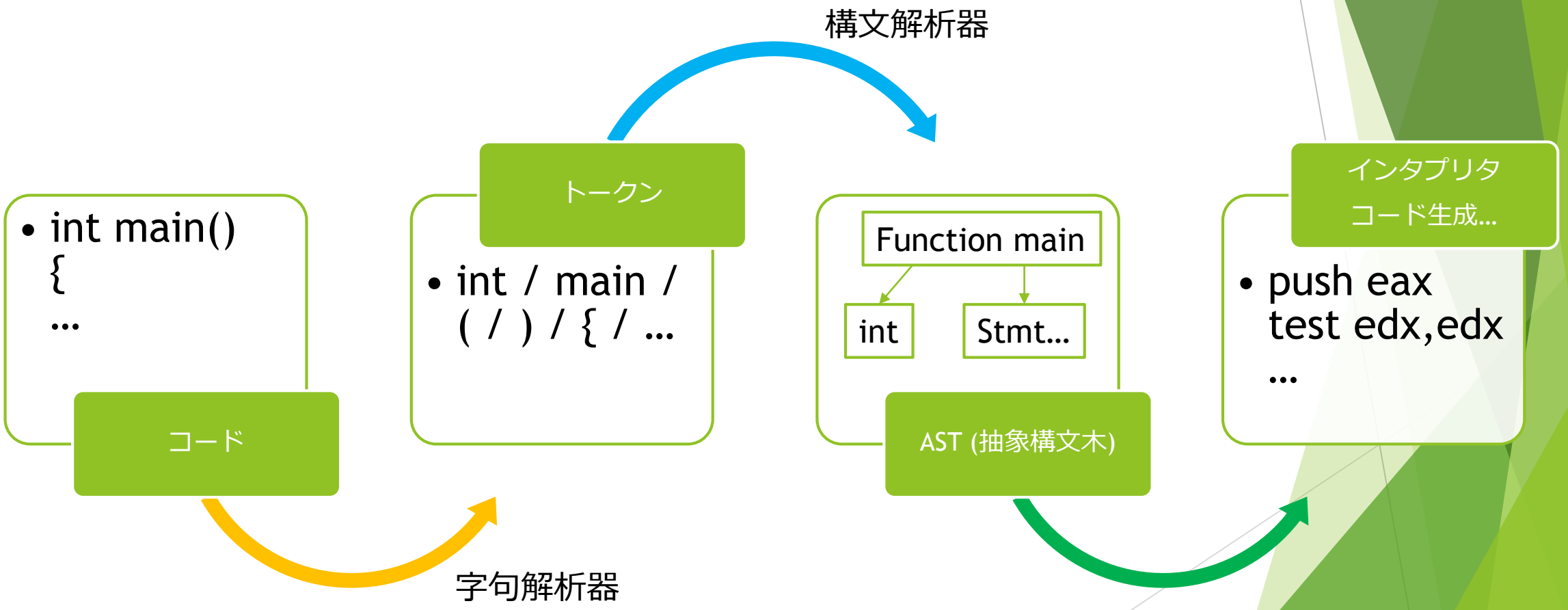
分科会の目標

- 簡単な言語を作る？
  - 簡単な言語のインタプリタを作る
  - バイナリ生成みたいなこわい＞ ＜ことはやらない

コンパイラの仕組み(イメージ)



コンパイラの仕組み(イメージ)



字句解析器 (Lexer)

- 与えられたソースコードを，トークン（意味の区切りの最小単位）ごとに分解する
- 具体的文法には立ち入らない
- だから，たとえば C で「a+++++b」がコンパイルできない
  - a ++ + ++ b としか解釈できないはず
  - だが，lexer は文法を知らないので a ++ ++ + b と解釈する
- 今回は lex を使う

構文解析器

- ソースコードの文法構造を明らかにする
- 解析木や抽象構文木を返して，プログラム的に扱いやすい形にコードを変換する

- 今回は yacc を使う

lex と yacc

- lex: 字句解析器を自動生成してくれる
- yacc: 構文解析器を自動生成してくれる
- 文法を記述したファイルから，C のコードを生成

- 現代では残念な仕様もある
  - データの受け渡しにグローバル変数を使っている (>_<)
  - 特に lex は C のコードしか出力できない (>_<) (>_<) (>_<)
  - (一応 yacc のほうは C++ を出力できるらしいけど…)
- それでも，(すくなくとも yacc は) Perl や Ruby の処理系を作るのに使われてたりするらしい

今回の目標

- とりあえず lex, yacc を使ってみる
  - 簡単な電卓みたいなものを作る

lex と yacc の使い方？

- 重要な部分だけ説明します
  - 「おまじない」は説明しようがない
  - http://kmaebashi.com/programmer/devlang/yacclex.html を見ましょう
- ここに，電卓の文法の例も書いてある

yacc

- yacc の入力は，定義部，規則部，ユーザー定義部からなる

- 定義部では，文法定義のために必要なもの（トークンとか）を定義する
- 規則部では，文法規則を具体的に記述する
- ユーザー定義部では，好きなことを書いていい
  - そのまま出力にコピーされる

定義部

- %{
  %}
  %union …
  %token …
  %type …
- %{ … %} には，C のコードを書ける
  - マクロなどを書く
- %left, %right みたいなものもある（あとで説明）

定義部

- %union では，構文を評価して得られた値を保持するための union を定義する
  - 得られる値は，整数かもしれないし，文字列かもしれない
  - まじめにパースする段階だと AST のノードへのポインタになるが，ポインタにもいろいろな種類がある
- たとえば，
  - ```
    %union {
      int int_value;
      double double_value;
    }
    ```
  - と書くと，整数と double を保持できる
    ```

定義部

- %token, %type では，記号を定義する
- %token <variable> … のようにすると，その記号の戻り値を共用体の variable に返す
  - ‘+’ を表すトークンのように，戻り値がない / 不要な場合は <variable> は省略可
  - %token は，lex から渡されるトークンを定義する
  - 定義しないで突然現れると怒られる
- %type は，内部的に使う記号を定義する
  - 戻り値を持たない場合は，%type で定義しなくてもよいらしい

宣言部

- ```
  expr : rule_1_1 rule_1_2 …
       | rule_2_1 rule_2_2 …
       …
       | rule_n_1 rule_n_2 …
       ;
  ```
  のような構文の羅列
- `rule_1_j` の数は何個でもいい
- また，マッチする条件を何個並列させてもいい
- `rule_i_1 …` の後には，`{ }` で囲んでアクションを伴える
  ```

宣言部

- この構文はどういう意味か？
- ```
  expr : term
       | expr ADD term
       | expr SUB term
       ;
  ```
  と言ったら，「式 expr とは，項 term または，expr + term または，expr – term」という意味になる
- 上の例からもわかるように，構文は再帰的定義が可能
- 例えば
  ```
  expr_list : expr
            | expr_list COMMA expr
            ;
  ```
  とすれば，expr の 1 個以上のカンマ区切りのリストが表現できる
  ```

アクション

- アクションは，その文法規則が実際に適用されたときに行う処理
- たとえば，expr ADD term を適用されたら，実際に足し算した値を返したい
  - 本当は，(左辺 + 右辺) を表す AST を返す必要がある
- ```
  {
    $$ = $1 + $3;
  }
  ```
  のように記述する
- $$ は，戻り値を格納する変数
- $i は，今の規則の i 番目を処理したときに得られた戻り値が入っている変数
  - expr ADD term だったら，$1 には expr の値，$3 には term の値が入っている
  ```

lex

- lex の入力は，定義部，規則部，ユーザー定義部からなる
  - が，yacc より簡単
- 今は規則部だけ書けば十分

規則部

- <token> <rule> といったものをたくさん書く
- <token> には，文字列（ex. “+”）または正規表現が書ける
- <rule> には，C コードを書く
  - 値を持たないトークンなら，return ADD; みたいにしておくだけ
  - 値を持つ場合（整数リテラルなど）は，{ } で囲んだ部分で値を代入する
  - <token> でマッチさせた文字列は，グローバル変数（>_<）yytext に入っている
  - 戻り値は，yacc の %union で定義したグローバル共用体（>_<）yylval に入れる
  - 値を持つ場合でも，return するのは yacc で定義したトークンの記号
    - 整数リテラルだからといって，読んだ整数を返したりしてはいけない

補足：演算子の結合性，優先順位

- http://kmaebashi.com/programmer/devlang/yacclex.html の例では，演算子の優先順位のために構造が若干複雑な入れ子になっている
  - 現実の言語では，優先順位はかなり複雑
  - expr, term やらの名前を考えるのも大変
- 演算子については，特に結合性と優先順位を指定できる
  - これを指定すれば，演算子の規則を 1 つにまとめられる
  - term はもういらない (expr だけで十分)

演算子の結合性，優先順位の指定方法

- ADD, SUB, … などの演算子を %token ではなく %left, %right で定義
  - %left だと左結合，%right だと右結合の演算子になる
- 優先順位の指定は，低いものを先に定義する
  - 同じ優先順位のものは，同じ %left / %right の中で定義する

練習問題

- http://kmaebashi.com/programmer/devlang/yacclex.html の電卓を拡張してみよう
  1. 他の演算子，例えば % (剰余) や ** (累乗) などを入れてみる
  2. 括弧 ( ) を入れてみる
  3. 整数表現を拡張してみる．例えば 0x123 みたいな表現をしたら 16 進数で解釈するようにする
  4. 演算子の優先順位の指定を使って，expr, term を 1 つにまとめてみる